unclassified
SECURITY CLASSIFICATION OF THIS PAGE

DTIC FILE COPY

2

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
AD-A217 682	5. MONITORING ORGANIZATION REPORT NUMBER(S)
702	
6a. NAME OF PERFORMING ORGANIZATION The Board of Trustees of the University of Illinois	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION AFOSR/NM	
6c. ADDRESS (City, State, and ZIP Code) 506 South Wright Street Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) AFOSR/NM Bldg. 410 Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER A~~FOSR~~ F49620-86-C-0136	
8c. ADDRESS (City, State, and ZIP Code) AFOSR/NM Bldg. 410 Bolling AFB, DC 20332-6448	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
61103D	3484	A5	

11. TITLE (Include Security Classification)
Supercomputer Environments

12. PERSONAL AUTHOR(S)
Da. Kuck

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 10/86 TO 10/89	January 9, 1990	33

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report provides a detailed summary of the Faust Project, a three-year program funded by the Air Force Office of Scientific Research targeted at the construction of an integrated parallel programming environment for the development of scientific and engineering applications.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Abraham Waksman	22b. TELEPHONE (Include Area Code) (202) 767-5027 — 22c. OFFICE SYMBOL NM

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

Center for
Supercomputing Research and Development

Final Project Report

Contract No. F49620-86-C-0136

Supercomputer Environments

Vincent A. Guarna, Jr.

January 9, 1990

University of Illinois at Urbana–Champaign
104 S. Wright Street
Urbana, Illinois 61801

REVIEW OF THE FAUST PROJECT

The objective of the Faust project is to provide users of high-performance parallel and vector machines with an environment that makes the task of programming easier. Included in this environment are tools that are used in the process of developing, testing, tuning, and using scientific and engineering programs. These include many tools that are specific to the problem of parallel programming such as restructuring compilers, parallel debuggers, and parallel program performance evaluation tools, as well as tools that are useful in any program development environment such as text editors, program management assistants, on-line documentation systems, and graphic visualization tools.

Philosophies

Developers of applications have varying levels of experience with respect to supercomputer usage. At one end of the spectrum is the "machine expert" who knows the intricacies of parallel and vector architectures and their effects on program performance and integrity. At the other end of the spectrum is the "applications expert" who has a background more focused on a specific scientific or engineering discipline. Both users have problems using high-performance architectures for the development of applications. The applications expert develops sequential programs that execute slowly, consuming excessive machine time. This user can benefit by the automatic parallelization and optimization tools supported by Faust. The machine expert is capable of achieving better performance but requires significant personal time to acquire the necessary data to debug and tune applications. This person can use the detailed instrumentation, visualization, and query tools supported by Faust to shorten the task of understanding program behavior. Faust's primary focus is to provide the development setting to support the development, debugging, and optimization tools for all users along this experience continuum.

Goals

The Faust project has been directed toward three major goals. The first goal is the design and implementation of a set of new tools aimed specifically at the problem of developing efficient scientific programs for supercomputers. This includes the creation of new interactive compilation tools as well as new facilities for debugging and performance evaluation in a parallel execution environment.

The second goal is portability. Although the Faust environment assumes the existence of a bitmapped workstation running Unix as a basic platform, Faust is expected to run on a variety of hardware. In order to accomplish this goal, all user interface libraries have been layered on top of the X Window System developed at MIT [ScGN88]. Additionally, all file operations have been designed to work on a single name space style of file system such as [illegible].

The third goal is integration. In addition to the development of new tools, Faust is intended to cooperate with existing program development tools such as system text editors and compilers without modification of those tools. A major focus for the Faust project is the definition of an architecture that supports this cooperation.

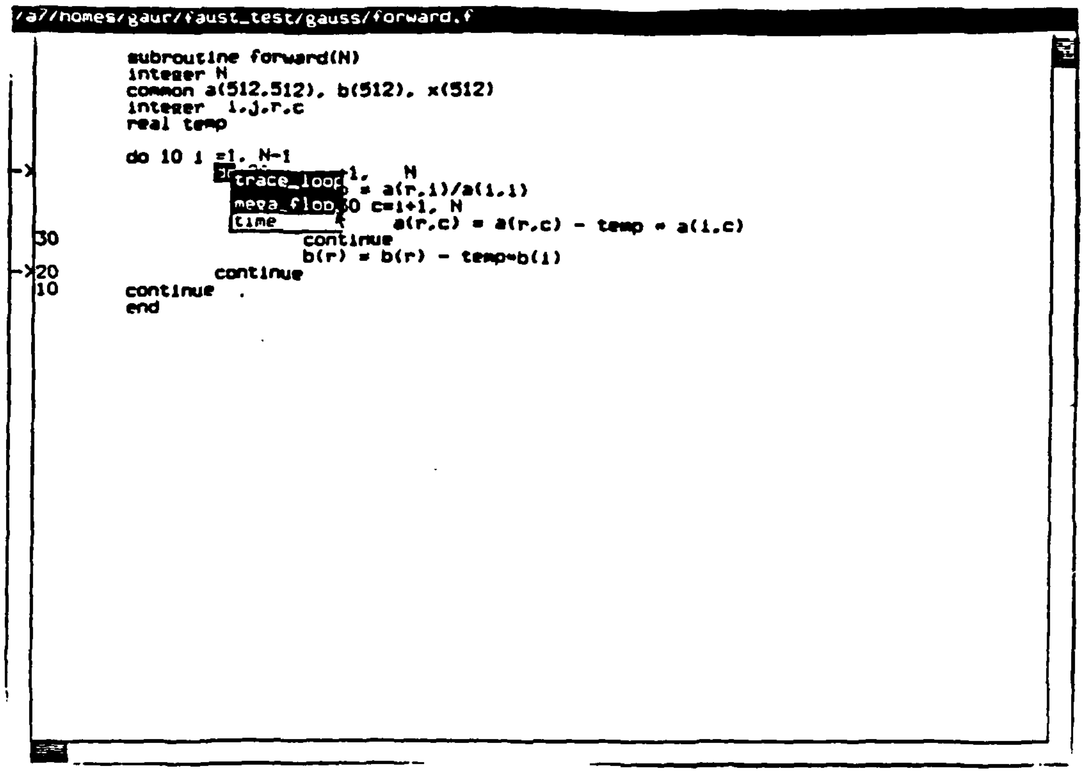
Integration

The concept of integration within the Faust environment is that of loosely-coupled, stand-alone components that are able to communicate through a shared collection of data called the Project Data Base. The integration supported by this data base is evident at two levels. At the lowest level, connection of tools to the data base allows the Faust environment to do many operations automatically that would otherwise be done manually by the user. In this context, integration provides a convenience to the user, reducing the number of explicit steps required to perform certain operations. An example of a tool that can benefit from this sort of integration is the Unix GPROF tool. GPROF is a performance profiling system that accomplishes its task by creating executable programs that leave a data file, gmon.out, in the directory from which the user runs the program. If the user wishes to execute the program multiple times, the data files must be saved and cataloged for later reference. Furthermore, if different executions of the program correspond to different versions of the program, the user's data file tracking job is more difficult. However, in this example, low-level integration can be introduced by associating the data files with the appropriate executables. In this context, the project data base provides the "glue" necessary to relieve the user of the burden of directory and file management. One aspect of the Faust philosophy is to provide exactly this type of integration.

A higher level of integration can be achieved when new tools are created. At this level, the overall functionality of two tools used in concert is greater than the functionality of the individual tools used in a stand-alone environment. At this level of integration, the goal is not simply user convenience but newer, more powerful functions that cannot be provided by either tool alone.

An example of this higher level of integration is the integrated performance experimentation tool. Using this tool, the user is able to request the collection of specific pieces of performance data. Figure 2 shows a sample interactive session where the user is requesting a report of the number of megaflops achieved during the execution of a particular Fortran DO loop. Note the "context-specific" menu that appeared when the DO keyword was clicked on. The high-level integration is at once apparent. From inside one tool, a simple mouse-based text editor, source language-specific behavior was achievable through the sharing of common data. In this case, row/column information known by the text tool was mapped into a particular syntactic construct by internal data structures generated by the Faust compilers.

To satisfy this particular request, two pieces of information are needed. First, a measure of the time elapsed during execution of the loop is needed. The second component is a count of the number of floating-point operations performed in the loop. Once this selection is made by the user, the experimentation tool issues requests to the program instrumentation tool to insert the "start timer" and "end timer" requests



```
/a7/homes/gaur/faust_test/gauss/forward.f

          subroutine forward(N)
          integer N
          common a(512,512), b(512), x(512)
          integer  i,j,r,c
          real temp

          do 10 i =1, N-1
                                 1,   N
                               = a(r,i)/a(i,i)
                               0 c=i+1, N
                                   a(r,c) = a(r,c) - temp * a(i,c)
30                      continue
                        b(r) = b(r) - temp*b(i)
20              continue
10        continue   .
          end
```

Figure 2. Loop selection automatically handled by the Text Manager

around the loop. The experimentation tool then queries the data base to determine if the number of floating-point operations can be determined statically (i.e., if the number of loop iterations is ascertainable at compile time). If so, no further modifications to the program (by Faust) are necessary. If not, the program must be additionally modified to count the number of iterations executed at run time.

The above sequence of steps are all examples of high-level integration. The automatic measurement of a particular program characteristic (megaflop rating for a loop) is accomplished through the cooperation of three Faust components, the text editor, the Fortran compiler, and the program instrumentation tool. As designed, none of these tools explicitly measure megaflops in user-specified loops. Moreover, none of these tools is capable of measuring megaflops in user-specified loops when operated individually. It is only by the cooperation of these tools through shared data that these higher level functions can be achieved. Creating the infrastructure for these high-level interactions is another focus for the integration goals of Faust.

Phase I Accomplishments

The first three years of Faust (Phase I) have been a period of foundation building, filling in many of the components needed to support the programming model presented in Figure 1. Many low-level tools were developed for the aid of the environment tool builder. Using these tools, some high-level tools were constructed. Below is a concise list of the accomplishments that will be completed by the end of the Faust funding period (September 30, 1989). They are organized into the five classes of problem spaces depicted in Figure 1. A more detailed discussion of each area follows:

PHYSICAL PROBLEM DOMAIN (Phase 1)

- PDE-tutor prototype developed
- Parallel ray-tracing, image generation developed.
- Matrix visualization tools developed

SOURCE CODE DOMAIN (Phase 2)

- Text handling tool developed.
- Abstract graph handling and display tool developed.
- Hierarchical program browser (graphical/textual) developed.
- Numerical subroutine library expert developed. Both conversation and graphical versions of on-line help systems have be prototyped to aid users in locating numerical library kernels for the Cedar system.

INTERNAL PROGRAM DOMAIN (Phase 3)

- SIGMA editor developed, supporting interactive restructuring and program queries for Cedar Fortran and Parallel C.
- Automatic error-analysis package integrated into Cedar Fortran compiler.
- Restructuring Lisp compiler developed (PARCEL).

PROGRAM DATA DOMAIN (Phase 4)

- Execution Analysis SYstem (EASY) developed. This is a facility for the automatic location of nondeterminism in parallel programs.
- Prototype of a parallel breakpoint debugger (XDBX) developed. Also developed

was a graphical front end for the debugger.

PHYSICAL MACHINE DOMAIN (Phase 5)

- Integrated performance experimentation environment developed. This tool will be supported in two forms. One, based on GPROF, will be a portable tool used for analyzing overall program performance at the application or subroutine level. The other, based on performance tools developed at CSRD will deliver more accurate profiling results as well as delivering new information to the user such as parallelism behavior and megaflop ratings for arbitrary sections of code.

- SIGMA editor extended to support queries for prediction of cache utilization and program performance.

Architecture

The organization of the Faust environment is shown in Figure 3. At the highest level in the system are the Faust user-level tools. These are the utility programs used by programmers to aid in the development of scientific programs. Included at this level are the traditional Unix program development tools such as system text editors and compilers, as well as Faust parallel programming aids such as the performance evaluation facility and interactive compilation tools. The lines between boxes indicate paths of communication between system elements.

Programmer tools access the file system through one of two paths. One is a direct path, involving no interaction with Faust facilities. In this model, programmer tools may be thought of as being integrated by virtue of their communication through side effects in the file system.

An additional path to the file system is supported through the use of the Project Manager (PM). The PM is a hierarchical data base manager used by Faust to create and manipulate the project database in a network-transparent manner.

For the user interface, Faust supports a layer of building blocks that is available to the Faust tool builder. The building blocks comprise interface utilities to do basic input-output operations with X Windows. In addition, the layer contains the mechanics to maintain a hierarchical program abstraction. This abstraction allows the user to view application programs in varying levels of detail, from a textual view of source code to a high-level graphical view of function and task relationships.

Low-level Tools

In order to unify the development of user tools for Faust, a set of low-level "building block" components were constructed to manage various abstract entities. These low-level tools manage textual and graphical objects for higher level tools to promote user interface and functional consistency as well as code reuse among tools.

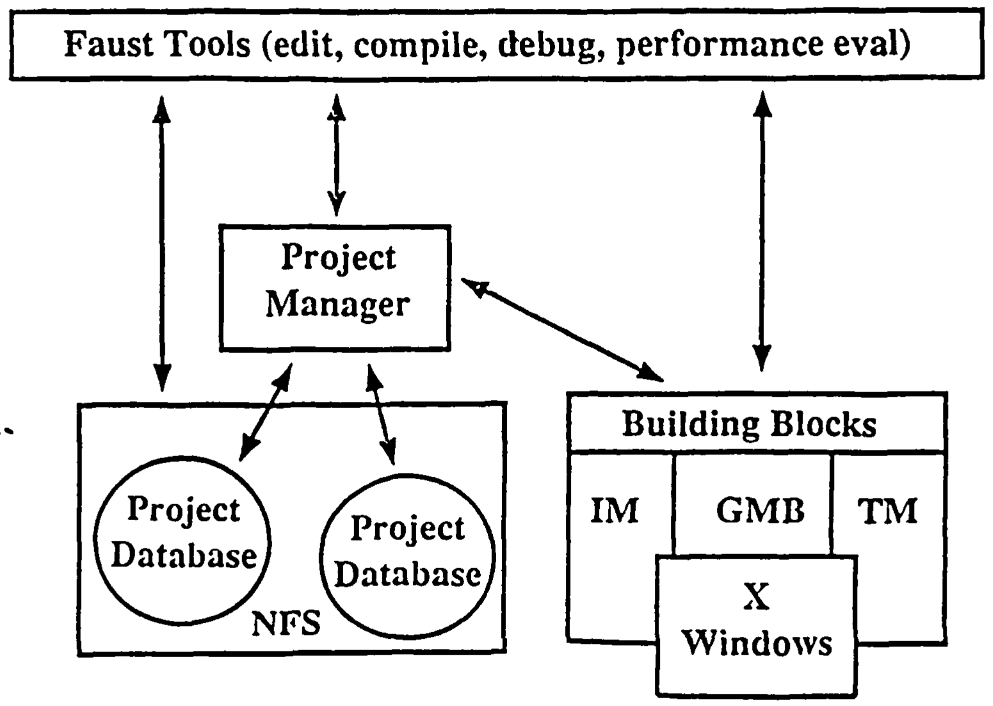


Figure 3. Architecture of the Faust environment

Graph Manager/Browser

Logically, GMB comprises two major components; the Graph Manager and the Graph Browser [JaGu89]. The Graph Manager is an abstract data type for graphs, providing full–featured support for graph manipulations including node addition/deletion and arc addition/deletion, as well as functions for attaching and retrieving an arbitrary number of user–definable data fields to and from each node and arc. The Graph Manager is meant to be strictly abstract in nature and accomplishes the sole function of manipulating graph data structures for application tasks without regard to associating screen representations with the data structures.

The Project Manager makes extensive use of the Graph Manager to store its data. Each proje[illegible] ultimately represented as a graph. Objects of the projects and relations between pr[illegible]ts are represented as graph nodes and arcs, respectively.

The G[illegible]h Browser is the user–interface of the graph tool. The Graph Browser's function is [illegible]use data structures constructed by the Graph Manager and perform a series of screen–mapping operations under application control. Attributes such as graph layout can be computed by either the application or the Graph Browser. The Graph Browser also supports the facility for mapping abstract graph node data into screen attributes such as color and visibility.

Development of a graph tool kit was justified for several reasons. Developers of packages such as compilers, debuggers, and performance tools frequently build internal data structures that are graph– or tree–like in nature. Additionally, by integrating a browsing tool with the graph tool, additional flexibility can be brought to the development environment. One side effect of having the integrated browser tool is that internal data structures can easily be presented to the user interface.

An emphasis of the Graph Browser was the realization of graph views and graph animation. The idea behind graph views is similar to the idea behind database views; the graph view contains only the subgraph of the original graph that is of immediate concern to the application or end–user. Once a graph view has been specified and is being displayed, the Graph Browser supports efficient animation of the graph. Animation of the graph includes changing node shapes, changing node and arc colors, changing node and arc text, and making arcs and nodes appear and disappear.

Another emphasis of the Graph Browser was automatic graph layout. Graph layouts that are hierarchical with a minimum number of edge crossings have been found to be most acceptable by end–users. However, the determination of optimal layouts with respect to these requirements have been shown to be NP–complete. [TiNe87] The Graph Browser uses its own set of layout heuristics that yield a polynomial–time layout algorithm.

Many efforts related to graph manipulation and display have been published in the last several years. Some of the work focuses on special types of graphs, such as trees or planar graphs. Other work, such as VLSI routing, is concerned with layouts of arbitrary graphs, but lacks the interactive flavor of GMB. For comparison, three recent research projects are examined that share GMB's focus for handling arbitrary graphs in an interactive framework.

GRAB, a graph browser implemented by Rowe, *et al* [RDMM87], concentrates on graph layout. The heuristics used are complex, consisting of multiple phases where at least one of the phases is iterative.

Tichy and Newbery implemented a prototype system called kb–edit (knowledge–based editor), which lead to the devolpment of a successor system, EDGE [TiNe87]. EDGE recognizes that very large graphs may need to be abstracted to be of use to an end–user. To this end, EDGE allows for the formation of node groups, and uses a simulation of three dimensions in two dimensions by stacking nodes on the display like a skewed deck of cards.

The ISI Grapher, a tool implemented by Robins [Robi87, Robi88], uses a linear–time graph layout algorithm. The ISI Grapher also concentrates on portability, versatility, and extensibility. The ISI Grapher has a potential shortcoming for displaying arbitrary graphs in that it breaks cycles by splitting a node into two nodes. This allows arbitrary graphs to be displayed as trees and results in a linear time algorithm. However, splitting one node into two may be counter intuitive to an end-user who is expecting a natural one-to-one correspondence between the nodes and the underlying objects they represent. In defense of the ISI Grapher, however, empirical evidence with GMB has shown that many graphs tend to be tree-like.

Animation of a graph is a useful feature. One of the primary goals of GMB was to support animated graphs. For animation, GMB supports changing colors, shapes of nodes, and visibility. Kb-edit, the ISI Grapher, and GRAB are primarily for static displays and support animation only at a primitive level. Kb-edit for example, supports greying of nodes for its animation.

Another unique feature of GMB is the support for multiple views where a view contains a subgraph of its parent graph. The ISI Grapher supports the displaying of only a subgraph of a specified graph, but does not support multiple simultaneous subgraphs. Neither kb-edit, EDGE, nor Grab mention any similar functionality. If a graph is used as an abstraction mechanism, then views are important to allow complex graphs to be considered from different abstract points of view.

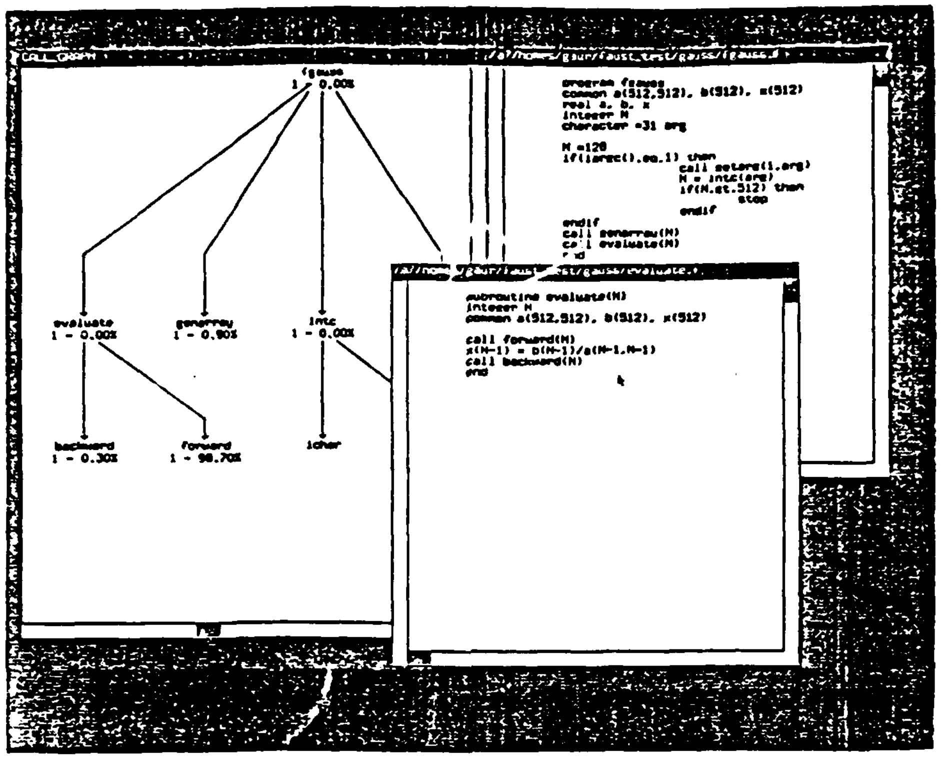
Text Manager

The Faust low-level tools include the Text Manager, a collection of routines for maintaining text objects in the system. The text handling subsystem provides an abstract data type for textual objects, including edit, search, and display functions. The rationale for including such a package in the Faust support layer is to promote reusability of basic functions to reduce the development time of higher-level tools.

In addition to supporting basic text operations, the Text Manager is coupled with the other support blocks to provide two higher-level functions. One is the graphical program browser. By using information stored in the project data base, the Text Manager can support a "zoom/unzoom" facility that allows the user to choose the level of detail at which to view the application. Selecting the unzoom function from a text manager window causes the creation of a subroutine call graph window to be displayed on the user's workstation. Similarly, the call graph window supports a zoom operation. The selection of a function or subroutine (by "clicking") results in the creation of a Text Manager window containing the source for the associated function. Figure 4 shows an example screen displaying the graphical and textual representations of an application program.

The other high-level facility support by the Text Manager is a source annotation facility. Source annotations are a way in which Faust tools can allow the user to place tool-specific "markers" in program source text. These markers are maintained in an external file (an annotation file) associated with the appropriate source file by the Project Manager.

An example of the source annotation facility is shown in Figure 2. The arrows in the left margin show that some annotations have been inserted by the user. In this case,



```
/a//homes/gaur/faust_test/gauss/fgauss.f

program fgauss
common a(512,512), b(512), x(512)
real a, b, x
integer N
character *31 arg

N =128
if(iargc().eq.1) then
                call getarg(1,arg)
                N = intc(arg)
                if(N.gt.512) then
                        stop
                endif
endif
call genarray(N)
call evaluate(N)
end
```

```
/a//homes/gaur/faust_test/gauss/evaluate.f

subroutine evaluate(N)
integer N
common a(512,512), b(512), x(512)

call forward(N)
x(N-1) = b(N-1)/a(N-1,N-1)
call backward(N)
end
```

Figure 4. Graphical and textual program browsing

the annotations are specific to the performance evaluation system and represent directives to the instrumentation package for monitoring options to be executed.

The pop–up menu shows the options that are available; this menu is configurable and dynamic. The particular menu that appears is a function of the lexical structure that is selected by the user with the mouse. To achieve this level of "understanding," the Text Manager uses the program data base to relate lexical tokens to semantic constructs. Once the construct is recognized, the annotation configuration file is consulted to select the appropriate menu for that construct. In the case of the performance tool, menus are defined for subroutine name selection, source line selection, loop selection, and variable selection. Once a selection is complete, it is recorded in the annotation file and later retrieved by the performance tools.

User Tools

The following sections describe currently supported tools that were developed as part of the first phase of the Faust Project.

Project Manager

The focal point for integration with the Faust environment is a hierarchical database system called the Project Manager (PM). The PM maintains a network–wide collection of constituent "objects" that are associated with an application program.

All user application work in the Faust environment is done in the context of *projects*. From a user perspective, a project roughly corresponds to an executable program, although the user has the ability to define a project's exact characteristics. The project is the unifying theme in the Faust architecture. It serves as the focal point for all tool interactions. Faust achieves its functional integration through operations on common datasets maintained in each project.

Several references concur on the use of a common object–based architecture for environment building [MuKl88, KaFe87]. Other research efforts have also recognized the utility of graphical representations to convey the structure of a project [AmOD88, RDMM87]. The Project Manager's primary distinction from other object database efforts is its handling of dependences and inconsistencies. Particuarly prominent is the PM's capability to resolve inconsistencies in parallel. This is especially useful in a scientific supercomputing environment where compilations, linkages, or user–level experiments may need to be carried out on different nodes in a heterogeneous environment. For homogeneous environments, the parallelization capability of the Project Manager is useful for reducing turnaround times for user work. Apollo Computer's DSEE product (Domain Software Engineering Environment [Apol85]) offers similar parallelization capabilities for program compilations; however, the Project Manager's object management paradigm is more generalized for arbitrary computation chores.

Database Components

There are several types of files that are routinely maintained by the PM for each Faust application. These include:

- *Executable files* – the ultimate target object in which the end–user is interested.
- *Source files (.f, .c)* – original program text written in Fortran or C.
- *Object files (.o)* – intermediate files generated by system compilers in the process of producing an executable program.
- *Assembler files (.s)* – assembly language versions of the source files produced by the system compilers. These are specifically created by the Project Manager for reference by the performance prediction tools.

- *Dependence files (.dep)* – symbol table and data dependence information collected by Faust compilers for reference by the query/interactive restructuring environment (SIGMA).
- *Program graph* – static call graph data used by the Faust graphical program browser.
- *Execution trace files* – collected at run time as a result of the user's application being instrumented by the performance evaluation tools. These trace files are referenced by performance analysis and visualization tools.
- *Annotation files* – contain detailed information about modifications applied to application programs on behalf of Faust tools. For example, an annotation file exists for each execution trace that is collected by the performance tools. The annotation file contains the detailed descriptions of the performance data collected for the associated trace as well as the purpose for doing the collection.

Figure 5 depicts a small project, written in C, with the hierarchical organization maintained by the PM.

A Graphical Makefile Tool

One program development tool that was constructed with the Faust building-block layer is a graphical version of the Unix "Make" utility [Feld84]. The graphical makefile editor allows the user to specify program dependences by graphically creating a directed graph (see Figure 6). At the root of the graph (tree) is the executable object to be created. The next level of the tree consists of all of the object files required to generate the executable. Each object file serves as the root of a subtree which defines all files necessary to generate it.

The graphical makefile editor highlights those executable files which are out-of-date or inconsistent to remind the user which recompilations need to be performed. The user may specify certain subtrees to be recompiled or allow the system to perform all necessary build operations. In Figure 6, the nodes corresponding to inconsistent objects are highlighted with a box around or through the object name.

The rationale for developing the graphical makefile editor is twofold. First, it serves as a nice demonstration of the interface utilities developed for Faust. Second, the nature of traditional makefiles is perceived as tedious, making the investigation of a graphical approach attractive. Although building makefiles through the graphical editing tool requires as much (if not more) effort than standard makefiles, the final product appears to be more intuitive, giving users a better understanding of the relationships in the program. Additionally, the graphical representation provides a convenient method for inspecting the state of the application with respect to object file consistencies.

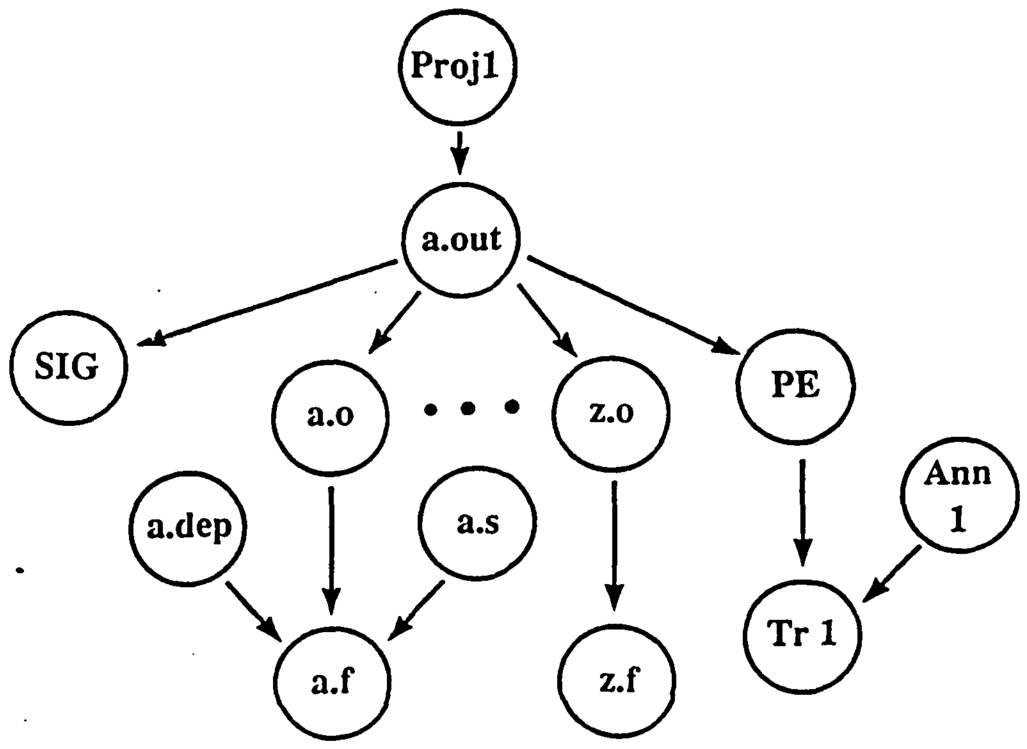


Figure 5. Example project database configuration

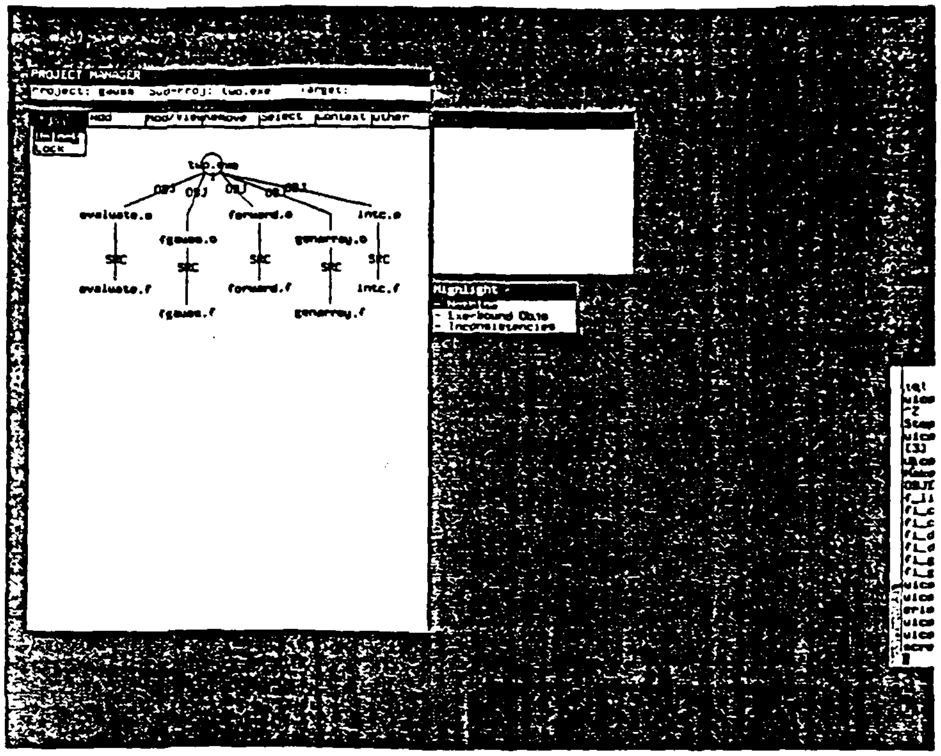


Figure 6. Graphical makefile editor

The SIGMA Editor

Much of what programmers do when they are trying to optimize a parallel program involves asking, and then answering, questions about the structure of the code. These questions can be simple such as "Why is this loop not executing in parallel?" to more complex questions such as "What is the impact of the machine memory hierarchy on the execution of this segment of code?" Just as interactive source code debuggers are essential in making software work correctly, special interactive tools are essential in answering parallel performance tuning questions.

Our approach to this task has been to try to merge the knowledge extracted from a parallelizing compiler with the knowledge collected by a program editor. The objective is a special family of programming tools that act as front–ends to a knowledge base of information about the target application. Our system is organized into six major components:

1. A parser for a parallel extension of FORTRAN (Cedar FORTRAN from CSRD in Urbana) with many 8X extensions.
2. A parser for a subset of C++ with vector and concurrency extensions.
3. A data dependence analyzer that builds a data dependence and control flow graph from the output of either parser. (Much of the dependence analysis for C programs was based on work done by Vince Guarna at CSRD [Guar87a, Guar88a].)
4. A database that resolves global flow information between the data dependence graphs generated from seperately compiled source modules.
5. A special library of parallelizing program transformations to aid the user in making the correct changes to the source code.
6. An interactive tool that provides the user an interface to the rest of the system.

Other research projects in universities and industry have also produced tools with similar goals. For example, the work of Bose at IBM on the EAVE system [Bose88] is based on the idea of building an interactive expert system for vectorization for the IBM 3090VF. EAVE incorporates heuristic rules for transforming loops into vector form and is based on techniques similar to our work on implementing an expert system back–end to the SIGMA system [Wang88]. The idea of an interactive global dependence database was introduced by Kennedy, Allen, Baumgartner and Porterfield in the PTOOL system [ABKP86]. FORGE from Pacific Sierra Research is a commercial package that interactively guides the user through a session with their vectorizer/parallelizer VAST. More recently, XYZ at Georgia Tech has developed a tool with more user–directed restructuring than VAST that is similar to some of the features in Sigma. The key difference between Sigma and the others is that it is designed around the concept of a multiwindow text editor as a front–end for a database of dependence and object code information.

To illustrate the ideas involved and how the system is used, we will list a few simple examples designed to show the nature of the information stored into the database by the compiler. These ideas are illustrated by the examples in the call to the subroutine Sub1 in the loop below:

```
real A(100,200), B(100,256)
do i = 1, n
    do j =    1,  m
        call  Sub1(i,j,A,B,C,m,n)
    enddo
enddo
```

From the perspective of parallelism, the programmer is most interested in asking questions like:

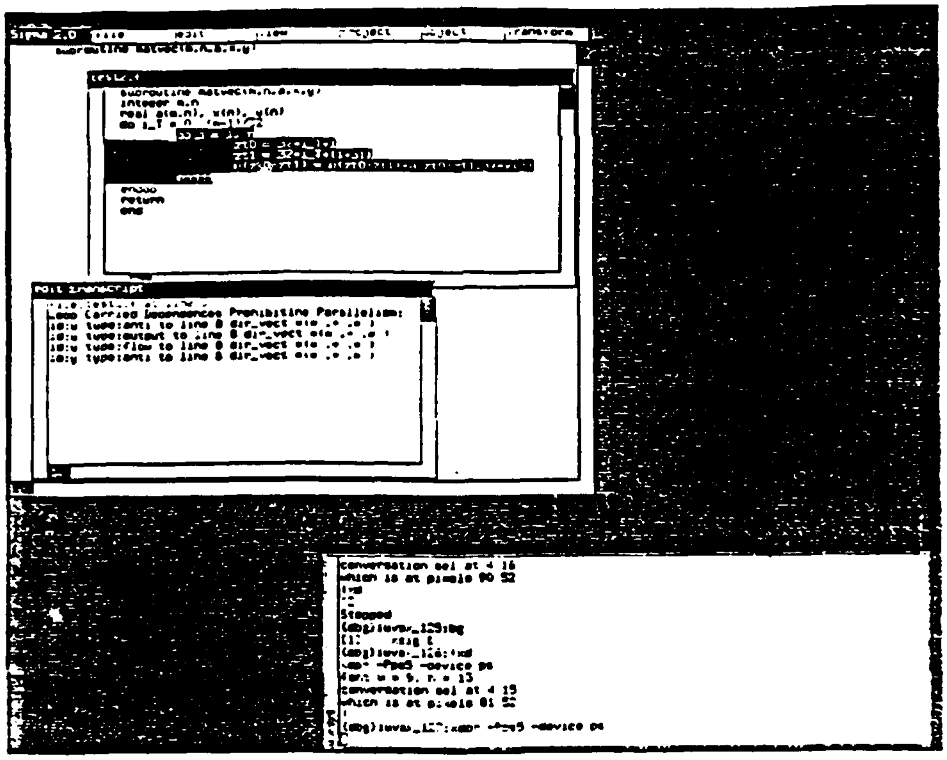
- Can the outer loop be executed in parallel? If not, why?
- Can the inner loop be parallelized?

- Can the loops be interchanged?

The answers to these questions depend on the way the subroutine `Sub1` (and the subroutines that it calls) uses and modifies its parameters and other external variables. The database also maintains this interprocedural analysis information, so more specific questions can be put to the system about the way variable values are used and modified. For example:

- For a given instance of `i` and `j`, what subsets of the arrays `A` and `B` are modified by the call to `Sub1`?
- What are the subsets of array elements accessed by a call to `Sub1`?
- Are there any common variables modified by a call to `Sub1`?

The answers to these questions should be posed as algebraic expressions in terms of the parameters and variables external to `Sub1`. For example, if `Sub1` takes the form of a C function:



```
subroutine matvec(m,n,a,x,y)
```

```
subroutine matvec(m,n,a,x,y)
integer m,n
real a(m,n), x(n), y(n)
do i_1 = [illegible]
      [illegible]
            [illegible]
            [illegible]
            [illegible]
      [illegible]
enddo
return
end
```

```
[illegible]
Loop Carried Dependences Prohibiting Parallelism:
10:y type:anti to line 8 dir_vect [illegible]
10:y type:output to line 8 dir_vect [illegible]
10:y type:flow to line 8 dir_vect [illegible]
10:y type:anti to line 8 dir_vect [illegible]
```

```
conversation sel at 4 16
which is at pixels 90 52
[illegible]
Stopped
(dbg)[illegible]
[1]   [illegible]
(dbg)[illegible]
[illegible] -device ps
[illegible] m = 5, n = 13
conversation sel at 4 15
which is at pixels 81 52
(dbg)[illegible] -device ps
```

Figure 7. An attempt to parallelize a loop using SIGMA

```
Sub1(i,j, A, B, C,m,n )
float **A, **B, *C;
int i,j,m,n;
{
int s,t;
for( s = i; s <= n; s = s+2){
     for( t = i; t <= m; t++){
          A[j][t-i+1] = A[j][t-i+1] + B[s][t]*C[t];
          }
     }
}
```

then the system will reply that a call to Sub1 modifies A[j][1:(m-i+1)] and uses B[i:n:2][i:m], C[i:m] and the same subset of A as well as the scalars i,j,m,n. Given these answers from the database, it is not hard to see how the system could decide that, in the nested loop with the call to Sub1, the outer "for s" loop is not parallelizable, but the inner "for t" loop is. In general, these are not easy questions to answer. A more complete theory on how to compute the access structure for arrays in iterative computations is given in [GaJG88a, GaJG88b].

In addition to answering questions about data dependences and the semantic structure of the program, the system can respond to requests to apply parallelizing transformations to the code. By using the mouse to select a segment of code in the text editor window, the user can ask to have any of a number of standard parallelizing operations listed in a menu applied directly to the body of the text. The system responds by observing where the text selection took place. Then it relates this line number in the source file to the point in the data dependence graph corresponding to the selection. The data dependence information is then used to decide if the selected transformation can be legally applied to the code without changing the semantics of the computation. If the answer is yes, the system generates text that corresponds to the transformed program and directly inserts the modified code into the source text to replace the transformed segment. For example, Figure 7 illustrates an attempt to apply parallelization transformation to the innner do loop. This parallelization is not possible because of the dependences given in the edit transcript window.

Performance Evaluation

Performance evaluation represents a high-level user tool in the Faust environment. The goal is to combine new facilities for collecting, analyzing, and visualizing performance with the tools in Faust so that the user's performance data is closely coupled with additional information in the environment [GJMY88]. Other systems have also attempted the integration of performance evaluation tools with programming environments, most notably the Carnegie Mellon's PIE environment [SeRu85] and the IPS system at Wisconsin-Madison [MiYa87]. In comparison with these systems, we have focused initially on improving the quality of the performance data collected and on analyzing the data to explain phenomena at all levels in a real system. Thus, our main accomplishments in the performance evaluation area have been in the building of performance tools. These tools, discussed below, are being integrated into the Faust environment.

Performance Data Collection

Performance measurement of parallel supercomputer systems necessitates data collection tools integrated with the machine hardware, system software, compilers, and user-level libraries. Unfortunately, consideration for performance measurement is relegated to after the machine has been designed and built in most systems. An exception is the hardware performance monitor on the Cray X/Y-MP systems [Lars86]. Unlike Cray, however, we are taking a broader approach that provides performance measurement services at all levels in the programmer's environment.

Hardware Monitoring

Monitoring of hardware events must tackle the problems of event detection, triggering, combination, and storage. We have built these fundamental components of a hardware performance monitor for capturing and analyzing hardware events [Lave89]. Basic modules are available for probing, event detection/combination, counting, and timing. These modules will be used to gather data about network and memory performance on Cedar. For the Alliant FX/8 machine, we have also employed a data acquisition system for gathering data about cache performance.

System-Level Measurement

At the system level, new techniques for timing parallel and multitasking programs on Cedar have been developed [BELM88]. Sampled-based timing approaches, used in the Alliant Concentrix OS, were replaced by state-measured timing. This allows high-resolution timing data to be collected for execution and nonexecution states on a per-processing-resource basis. Process context switch tracing has also been added as part of the system-level data collection. This has allowed accurate timing information to be recovered from fine-grained program tracing (see performance analysis section). The timing techniques developed here can be applied directly to other shared-memory multiprocessor systems.

Instrumentation has also been placed in Concentrix to monitor a program's utilization of concurrency resources on the Alliant FX/8. The measurement is sample-based with a snapshot of the concurrent execution state of the Alliant's processor complex taken at every sample period. A histogram of concurrent activity is collected over the program's execution lifetime.

Compiler–Level Measurement

Data collection at the compiler level includes mechanisms for subroutine profiling based on Unix GPROF techniques [GrKM82]. There are limitations of this approach for performance measurement of parallel programs and new profiling approaches based on tracing have been implemented (see below).

An automatic object code instrumentation tool, called LEECH, has been built to patch a user's program at the object code level with code for data collection; the concept of LEECH came from work done at Encore on the Parasight tool [ArGe88a, ArGe88b]. LEECH can be regarded as a post–processor of a user's program in that it works from an instrumentation specification describing where and how the code should be modified. The goal of developing LEECH was to remove the need for manual source code instrumentation. It also allows support for certain types of instrumentation previously inaccessible to the Cedar Fortran compiler. In particular, LEECH is able to instrument for concurrency instructions provided by the Alliant hardware. More generally, because of

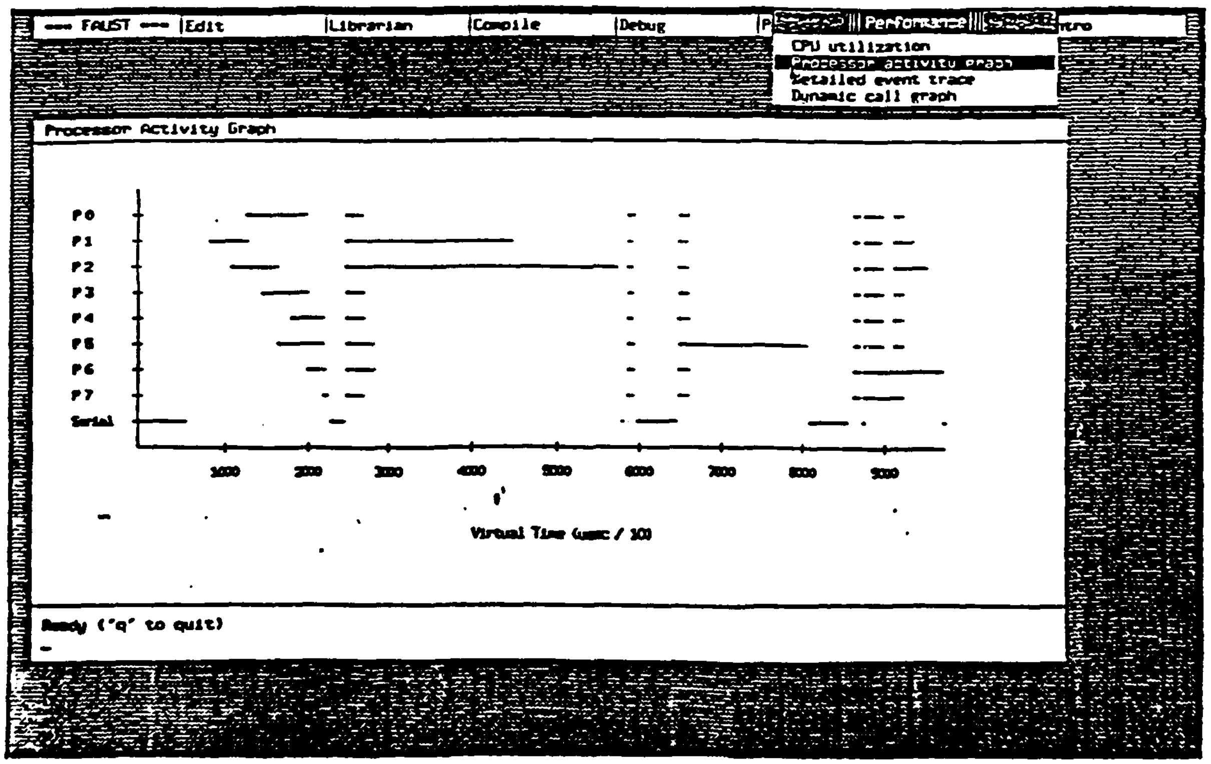


Figure 8. Processor activity graph

its modular design, versions of the LEECH can be easily generated for other hardware platforms. For example, a version of LEECH has been implemented for the Sun 3/50 workstation.

User–Level Data Collection

User–level data collection libraries have been built to support counting, timing and tracing of parallel, multitasking programs [Malo88]. These tools exist as routines and run–time support for data capture, analysis, and storage. The counting routines count user–define program events. The timing library interacts with the system–level timing facility to gather time samples on a per–event basis. The tracing library supports the tracing of program events. Each event is recorded in a processor trace buffer with a 10 microsecond timestamp of when the event occurred. The context switch tracing at the system level allows accurate timing information to be recovered from the timestamped

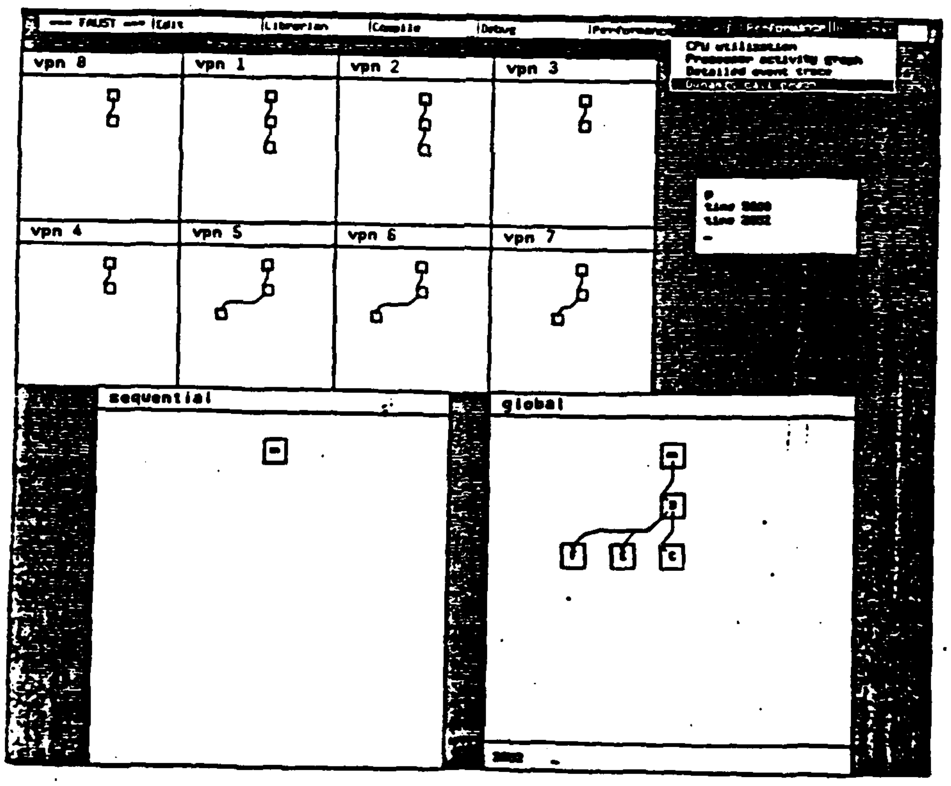


Figure 9. Dynamic call graph

trace. These user-level libraries are portable to any multiprocessor systems providing some form of timestamp generation.

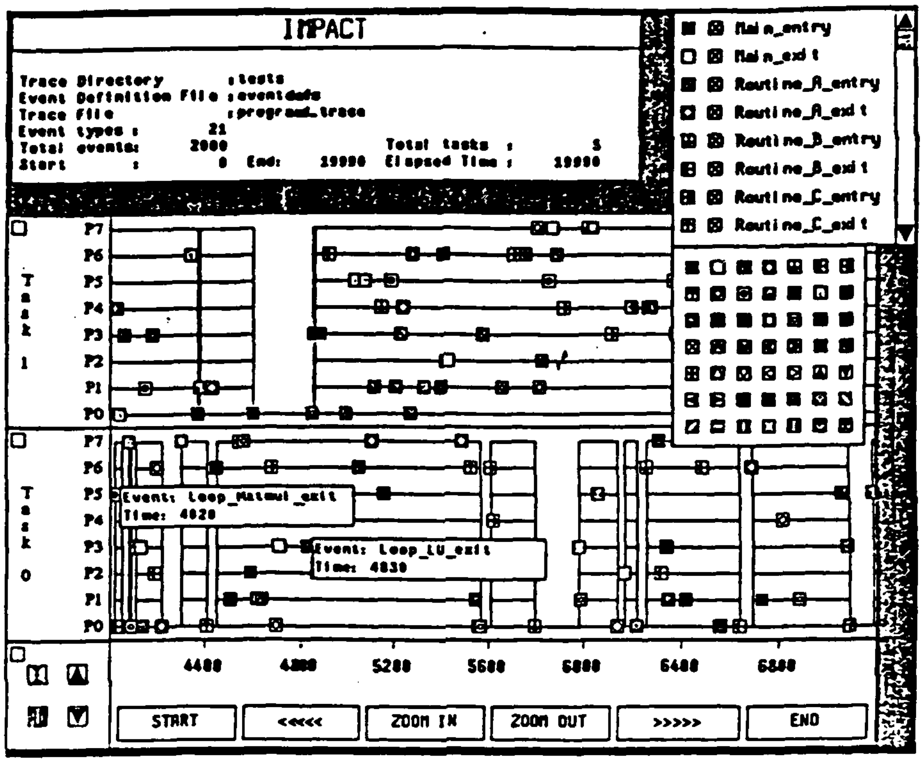
Performance Interpretation/Visualization

The goal of performance interpretation/visualization is to condense the potentially large amount of performance data into meaningful representations that elide irrelevant details. We have developed several tools that are useful in abstracting performance information into forms that more closely fit conceptual user models of performance behavior.

processor activity graph: The processor activity graph shows a timeline indicating the state of activity (active, inactive) for each processor participating in a program's execution (see Figure 8). The user can visibly observe from the graph time periods where there are low levels of parallelism. For instance, the figure shows full parallel activity between times 2500 and 2700 but low parallelism overall. In particular, the period 3000 to 5000 has two or less processors active.

dynamic call graph: Another abstraction of program execution is the routine call graph. We used the program event trace as an execution history from which the currently active (parallel) paths through the call graph can be calculated at any time during program execution. At that time, these paths are shown graphically as a subset of the entire call graph for each active processor. Replay of program execution shows a dynamically changing call graph reflecting routine operation behavior. For instance, Figure 9 shows the call graph state of the same parallel program execution as shown in the processor activity graph above at time 2652. All processors are active, as reflected by the call graph display in each processor window. The sequential window shows the routine in the program that invoked the concurrent execution. The global window shows all currently active paths in the program's call graph.

event display tool: A tool for graphically displaying events from multitasking programs has also been developed. The goal is to allow the user to visibly observe all or some of the events generated by all program tasks within any time window during program execution. Displays from several concurrent tasks can be presented simultaneously (see Figure 10). Sequential and concurrent operation is shown for each task by lines of processor activity. Events are shown as graphic icons in the displays. The user can interactively select which events are to be viewed and what graphic icons are assigned to events. Clicking on an event icon allows more detailed information about the event to be seen. The event display tool design draws from ideas found in the Graphical Multitasking Analysis Tool (GMAT) from Lawrence Livermore National Laboratory [SCSS88] and from BBN's GIST tool for the Butterfly multiprocessor [BBN88].



```
Trace Directory        :tests
Event Definition File  :eventdefs
Trace File             :program_trace
Event types :          21
Total events:        2000                  Total tasks :         5
Start        :          0  End:     19990  Elapsed Time :     19990
```

Event: Loop_Matmul_exit
Time: 4020

Event: Loop_LU_exit
Time: 4030

Figure 10. Event trace time line

Integration

The above performance evaluation tools and facilities are being integrated into the Faust environment along several lines. As mentioned earlier, the output produced by the data collection tools will be available to the Project Manager (PM) for storage in the hierarchical project database. Once a part of the database, the user will be able to use the PM facilities for maintaining performance results from multiple performance experiments.

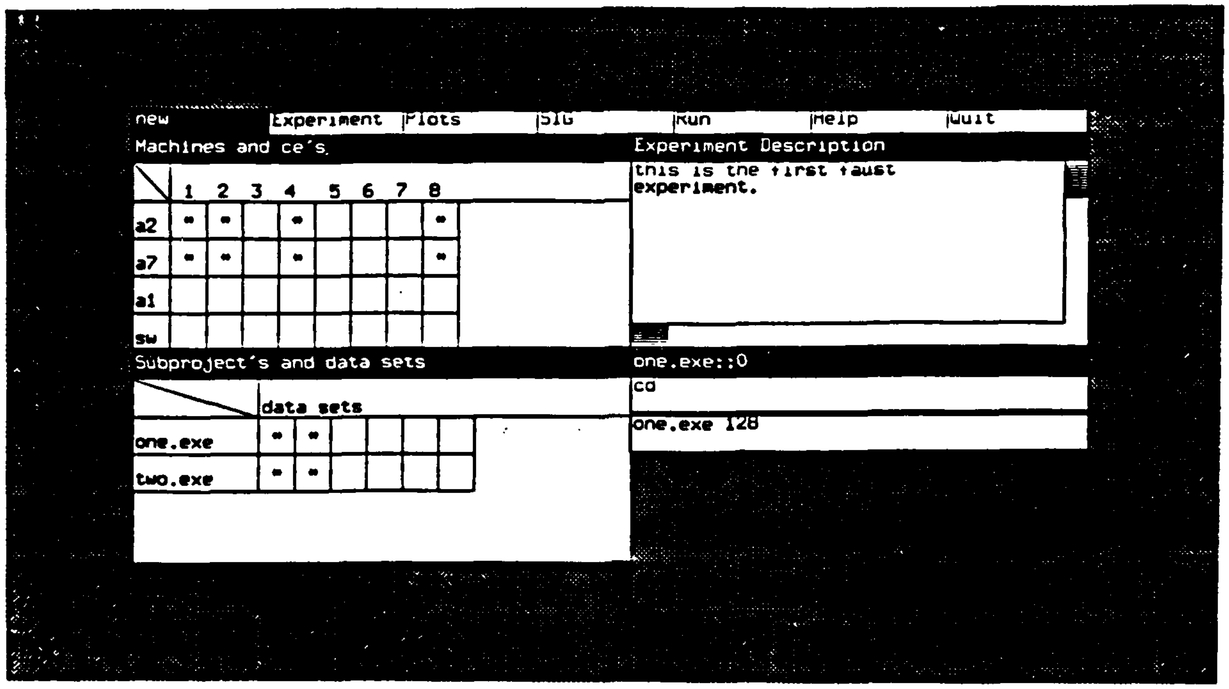
LEECH will become the standard tool for program instrumentation. An interface for specifying where and what instrumentation should be done will be developed and made a part of Faust's support for program interaction. This interface will also make use of the program database generated by the SIGMA editor and system compilers.

An interface to the performance analysis tools will be implemented that uses the Project Manager for gaining access to performance data files in the project database. Other information needed by the tools, e.g. program event descriptions, will also be accessible throught the database.

Finally, the performance visualization tools will be brought under the Faust umbrella so that access to other information maintained by Faust will be available while performance data is being viewed. Questions about source code, data and control dependences, or algorithms are all inquiries a user might make when studying performance information.

Performance Experimentation

The initial version of the Faust Programming environment supports a portable facility for using system performance evaluation tools and the Faust project management tools in a single, integrated data collection and experimentation environment. The data collection phase uses the GPROF facility supplied as part of the standard



	1	2	3	4	5	6	7	8
a2	•	•		•				•
a7	•	•		•				•
a1								
sw								

	data sets					
one.exe	•	•				
two.exe	•	•				

this is the first faust experiment.

one.exe::0

co

one.exe 128

Figure 11. Experiment definition in the automatic GPROF tool

programmer's workbench with the Unix operating system.

To conduct an "experiment" using Faust, the user first uses the Project Manager (PM). While in the PM, the user informs Faust of the location of the series of Fortran or C source files that comprise a particular application. The location of these files may be widely distributed as long as they can all be uniformly referenced through NFS. Once the PM has been given the list of files, it automatically constructs an *application subproject.* The subproject is a list of application components such as source files, object files, and executable files, as well as their inherent relationships, much like the relationships defined in a typical Unix makefile. However, the Faust subproject contains additional components such as symbol tables, dependence graphs, assembler files, and performance data files in order to support the various tools in the environment.

Once the subproject has been created, the user invokes the GPROF environment from the main menu. The GPROF experimentation environment allows the user to specify a series of parameters that are to be used when running the experiment. These parameters include the algorithm or algorithms to be tested, data files on which the algorithms are to be applied, one or more target machines, and the number of processors to be used for each machine. Figure 11 shows an example of an experiment description in the GPROF experimentation environment. In this example, the user is requesting that a single program be run on two different Alliant machines. For machine "a2," the program will be run a total of four times — once with one processor, once with two, once with four, and once with eight. Similarly, the application program will also be run four times on machine "a7" with the same processor configurations.

Once the parameters have been defined, the experiment can be run automatically. When the "run" command is issued, the GPROF tool comminucates with the Project Manager to locate all source files necessary to build an executable module of the application program. Once the sources are located, the Project Manager initiates the compilation of the application program for each machine that will be involved in the experiment. These compilations are done with the
-pg option that causes system compilers to generate executable modules that create the GPROF information at run time.

After the executable files have been created, the Project Manager runs the designated program in the configurations specified by the user. The *gmon.out* files that are created are subsequently scanned, condensed, and cataloged as a constituent component of the applications subproject. After all executions have been performed, the user may view the results of the experiment.

Experimental results make be reviewed in two ways; both are shown in Figure 12. On the left of Figure 12 is a subroutine call graph that has been annotated with GPROF data. Each node in the graph corresponds to a subroutine and has two numbers associated with it: on the left is the number of times the subroutine was called during execution, and on the right is the percentage of time the application spent in that subroutine.

The right part of Figure 12 shows a graph plotting speedup of an application versus processors for results on machines a2 and a7 together with an ideal speedup. While

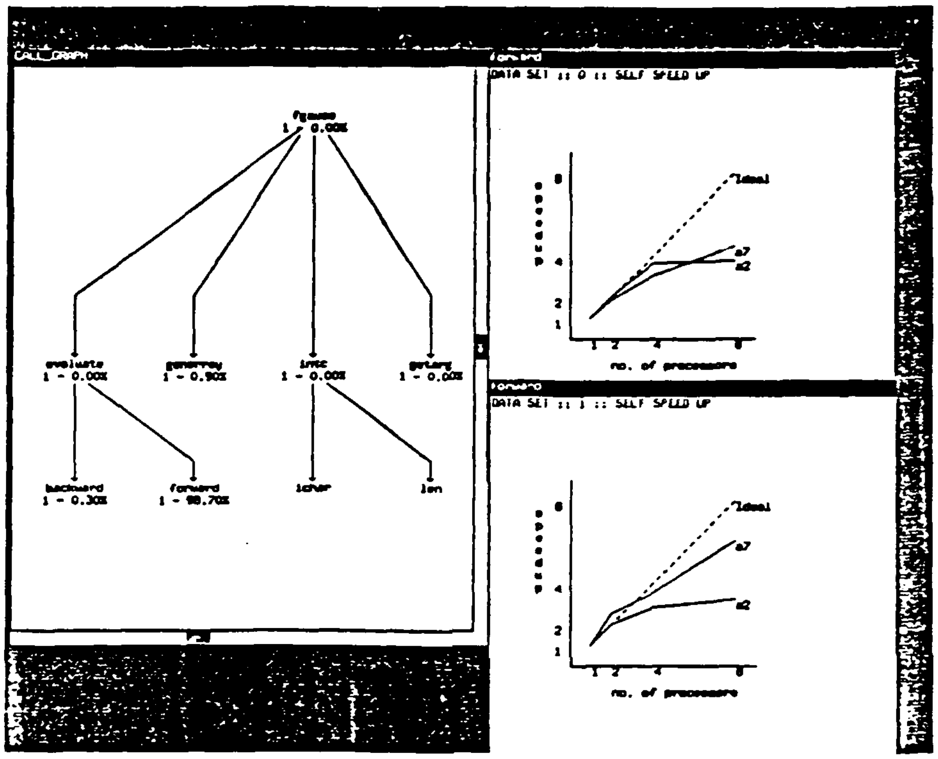


Figure 12. Experiment results from automatic GPROF tool

reviewing experimental data, the user can select the curves to be plotted on the same set of axes. Additionally, the data can be viewed at the application level or subroutine level.

Graphics

We have developed a high-speed, high-quality parallel ray tracing program (VRT) targeted specifically at the visualization of scientific data. VRT supports a complete illumination model including multiple-colored light sources, shadows, reflections, transparent surfaces, texture mapping and adaptive anti-aliasing. Special input primitives such as meshes and scalar 3-dimensional volumes are included to directly render results from computational fluid dynamics and other applications. Together these features are easily combined to render surfaces of constant density in a flow field with mirrors strategically placed to allow the researcher to see several views of the field in one image. Shadows cast on the floor provide additional three-dimensional information. The transparency allows several layers of iso–surfaces to be usefully rendered in the same image.

VRT uses an adaptive octree data structure [Glas84] to reduce the number of ray–object intersection tests from the classical O(R*n) to O(R*log n) where n is the number of objects in the scene and R is the number of rays generated (always greater than the image resolution, and typically in the range of 10**5 to 10**7). Although the octree reduces the effectiveness of vectorization by storing the object data in a nonlinear structure, the resulting savings in intersection tests is 20– to 200–fold in typical scenes.

Running VRT on an Alliant FX/8, we have rendered 32,768 polygons into a 512 x 512 image with 24 bits of color and with anti–aliasing in 20 minutes, plus another 20 minutes of preprocessing time. A good image with this many polygons requires extensive anti–aliasing since each polygon is so small. However, with this feature disabled, the same image is produced in about 3.6 minutes. VRT realizes good concurrency with a speedup of about 5.3 times on eight processors as compared to a single processor. The processing time to produce an image is directly proportional to the resolution of the image, with a 512 x 512 image taking nearly 4 times the processing time of a 256 x 256 image. With antialiasing enabled, this is reduced to just over three times, with the smaller image requiring additional samples to produce a smoother image. There are no standard benchmarks for ray tracing programs. However, Eric Haines [Hain87] has proposed some standard databases to be used when reporting results. Using the "mountain" benchmark on an Alliant FX/8, VRT renders 8,192 polygons and four glass spheres in 11.5 minutes (512 x 512 image, no antialiasing). Haines' conditions require VRT to do extra work by explicitly disallowing certain optimizations such as tree depth pruning that VRT normally applies.

Within the last year, graphics researchers have developed methods of directly viewing a three–dimensional volume of scalar data. Rather than extract isovalued surfaces to be rendered as polygons, the data is treated as a kind of fog or cloud. The data are passed through a function which may modify values, assign them pseudocolors, or eliminate values outside a specified range. The function may also provide a nonlinear mapping of the data. The resulting values provide the "density" of a fog to be rendered [UpKe88, DrCH88]. The resulting image may appear confusing when viewed as a static two–dimensional view, but often becomes very clear when a short animation is created showing the cloud rotate.

VRT has been enhanced to include this type of volume visualization. This work has advanced the state–of–the–art by including scalar volume as a primitive data type in the ray tracer. Ours is the only rendering program (of any type) that we know of which incorporates volumes with other primitives for display. To the user, it appears as just another primitive. But, now we can embed polygons in the volume as part of the rendering. For example, a realistic metal airplane wing can be embedded in a fluid flow or a grid set into a field of electrical potential. The fog nature of the images causes an attenuation of the image proportional to the density of the fog. Also, the volume can be viewed with reflections and shadows. By placing the image near the corner of a "room" with mirrored walls, we can see three views at once [illegible] the front and reflections from two

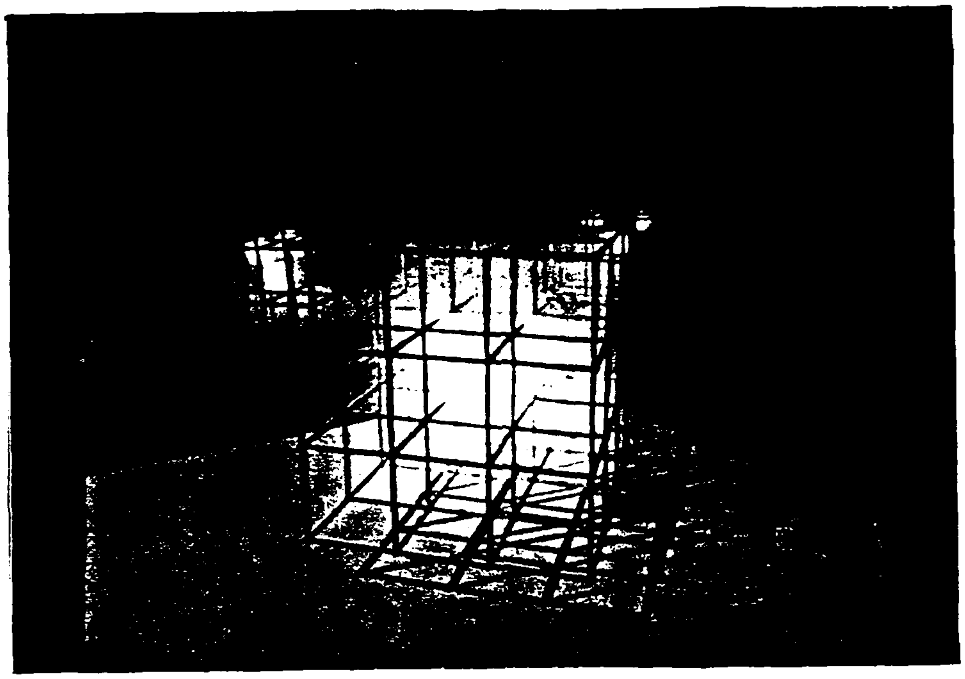


Figure 13. Three–dimensional field of scalar data rendered by ray–tracing

side angles. Figure 13 shows a scalar field of electrical potential using these techniques.

This provides more insight in a single image, but also allows the user to see the entire model in fewer animation frames. We are currently wrapping the program up for distribution to other sites for production or experimental use.

Another area we have explored is that of the division of labor between a supercomputer and a graphics workstation. Other groups have applied distributed processing techniques for visualizing results of supercomputer applications on workstations [WCHW87, JHHR88]. They have generally treated the workstation as a "dumb" frame buffer with the supercomputer reducing the model to an image, or have depended on ultra–high–speed interfaces from the supercomputer to the workstation. In the former cases, bandwidth is a significant issue. We looked into this problem to try to identify an appropriate model for bandwidth–limited distributed graphics processing. We studied and implemented a graphics interface for a structural dynamics application running on an Alliant FX/8. We developed a general methodology and approach to this problem, and implemented the necessary tools, some of which are necessarily application program specific. Our final report on this effort is documented in CSRD Report 859 [NeTu89].

We began to investigate data structure visualization. Initial efforts have been applied to numerical linear algebra using color computer graphics to gain insights into algorithm behavior. The insights were used to design more efficient numerical algorithms for supercomputers. We developed MatVu for Matrix Visualization. In MatVu, color or grey level is used to show the static structure of the matrix, while a combination of color, highlighting, and animation is used to reveal the active portions of the matrix. The graphics technology is quite mature, but this application of computer graphics has not been well exploited. Dongarra and Sorenson have demonstrated MAP (Matrix Access Patterns) which displays access patterns within a matrix. With this program they address another aspect of visualization. MAP is a performance evaluation tool rather than an algorithm development tool.

A new and interesting discovery from this effort is the eventual convergence of the Jacobi iteration matrix to a preliminary block diagonal form in the presence of clustered spectra (eigenvalues or singular values which are extremely close in value). Having gained insight into the apparent block diagonal form using MatVu, numerical analysts then instrumented the numerical detection of this optimal form in the one-sized Jacobi algorithm. This numerical decoupling property insures not only immediate parallelism for multiprocessor computer systems but also a significant reduction in the total number of floating-point operations that must be performed, i.e. a lower algorithmic complexity. For machines having a hierarchical memory architecture (e.g., Alliant FX/8, Cray 2, CEDAR), the decoupling also yields greatly improved data locality in that the submatrices associated with the independent smaller-order problems may be stored and operated upon completely within the fast local (cache) memories. A report on this activity has been submitted as an article to IEEE Computer and appears as a CSRD Report 826 [TuBe89].

Restructuring Lisp Compiler (PARCEL)

We have completed the first version (1.0) of the Parcel compiler and run-time system. This system consists of a machine-independent, parallelizing compiler for Scheme, that produces from a sequential Scheme program a parallel object code in a machine-independent intermediate form, and a code generator and run-time system for an Alliant FX/8 running the Xylem operating system (the operating system of Cedar). The restructuring compiler can be used in a variety of modes. Of course, it can be used in the production of an image to be executed on the Alliant. Alternatively, the user may produce from the compiler a restructured version of his program, rendered in a source-level form that is a variant of Scheme, with annotations for parallelism. This mode of compilation might be used to assess the parallelism of an algorithm, or to anticipate the performance of the object code produced by the compiler, for example. Finally, a mode is provided whereby the user may obtain a view of the restructuring process itself. In this mode, the compiler emits "snapshots" of the program as restructuring proceeds.

These snapshots are produced at sufficiently close intervals so that the restructuring process is made quite apparent.

The parcel run–time system consists of a parallel stop–and–copy garbage collector, a library of parallel recurrence solution routines, and run–time support for parallelism introduced by the compiler, in addition to the facilities normally provided by a (sequential) lisp implementation (e.g., input–output). The system is being augmented to facilitate debugging and performance evaluation, and to accommodate a richer variety of data types. As we port it to Cedar, we are experimenting with a variety of strategies for the management of a hierarchical memory. This includes strategies for the parallel allocation and deallocation of objects in the setting of a global, interleaved memory, as well as the use of compile–time analysis for the placement of objects within cluster memory where their lifetimes permit. We are also experimenting with various strategies for the management of microtasks.

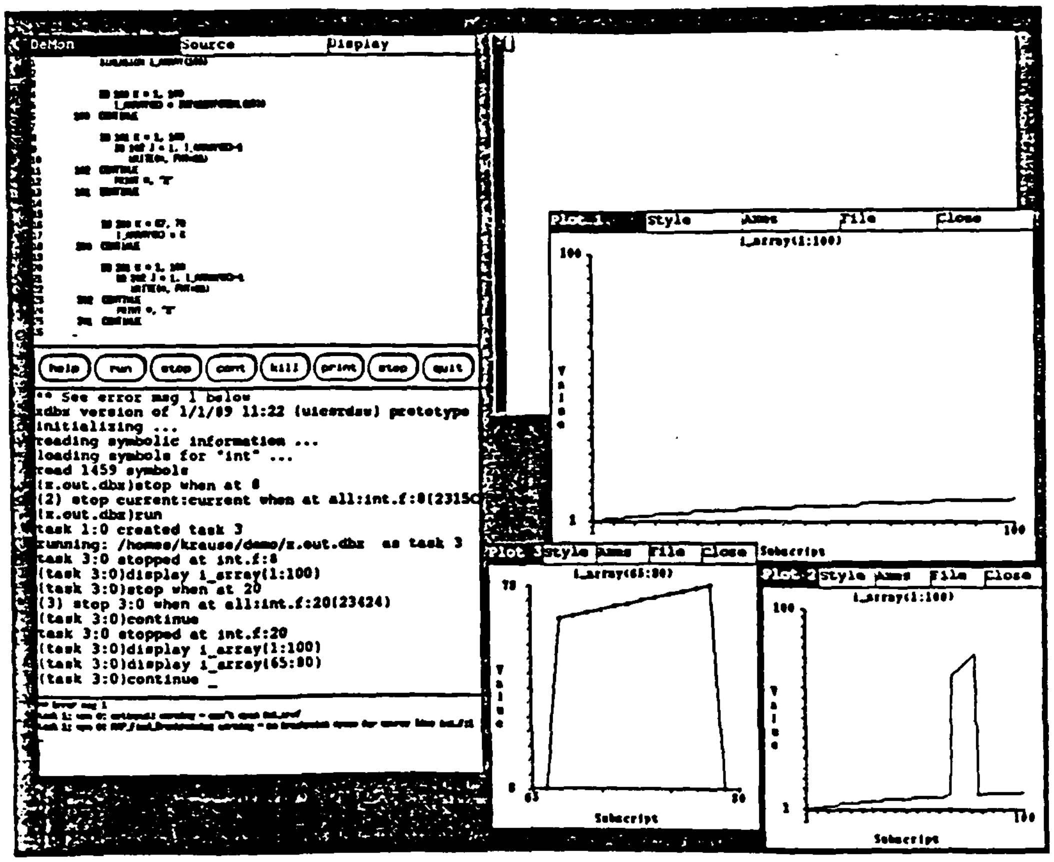
Debugging

All programming environments provide debugging support, either *de facto* via print statements in the program being debugged, or via a set of debugging tools. Given the complex interactions of parallel systems, parallel program debugging support is particularly important. Moreover, a parallel program debugger must *efficiently* monitor the computation state, lest the potential advantages of parallelism be lost.

Monitoring, no matter how unobtrusive, introduces perturbations in either the actual or the perceived partial order of computation states. In most parallel systems, including Cedar, these perturbations can create significant changes in the program's final state and can mask errors from the debugger (e.g., monitoring program execution to detect synchronization errors may change program behavior sufficiently to mask those errors).

Traditional, breakpoint debuggers (e.g., Unix DBX) require operating system intervention both to trace variable references and to examine the program state. As a consequence, most such debuggers greatly perturb application performance. For sequential programs, this is manifest as increased execution time — up to three orders of magnitude for variable tracing. In contrast, the perturbations in parallel programs include not only increased execution time, but also changes in the execution path. The computation requirements of most parallel programs make the increase in execution time unacceptable, and the nondeterministic execution makes debugging impossible.

Data dependence analysis provides the information needed to reduce debugger intrusion to manageable levels. By identifying program source code locations where changes to variables potentially occur, a debugger can minimize the program instrumentation necessary to detect user–specified conditions. To test feasibility of this approach, we have developed the XDBX debugger for Cedar Fortran that uses data dependency information to generate instrumentation. Although debugger functionality cannot be realized without some performance perturbations, analysis shows that the perturbations



```
** See error msg 1 below
xdbx version of 1/1/89 11:22 (uicsrdsu) prototype
initializing ...
reading symbolic information ...
loading symbols for "int" ...
read 1459 symbols
(x.out.dbx)stop when at 8
(2) stop current:current when at all:int.f:8(2315C
(x.out.dbx)run
task 1:0 created task 3
running: /homes/krause/demo/x.out.dbx  as task 3
task 3:0 stopped at int.f:8
(task 3:0)display i_array(1:100)
(task 3:0)stop when at 20
(3) stop 3:0 when at all:int.f:20(23424)
(task 3:0)continue
task 3:0 stopped at int.f:20
(task 3:0)display i_array(1:100)
(task 3:0)display i_array(65:80)
(task 3:0)continue _
```

Figure 14. Sample session with XDBX and graphical front–end

are minimal, making breakpoint debugging of parallel programs feasible.

Finally, we have developed an interface, based on X Windows, that supports mouse–based, graphical interaction with the debugger (see Figure 14). In addition, the interface supports the display of two–dimensional cross sections of Fortran arrays. Once an array is displayed, it can be combined with other array displays to show the difference between two data sets or overlaid data sets. This ability to display array cross sections and apply simple transformations is the essence of the visualization debugging of Section 3.

Automatic Nondeterminism Detection

We believe that automatic detection of the causes of nondeterminacy in parallel programs would be quite helpful, since timing problems are hard to detect and isolate with a conventional debugger. Toward this goal, work has been done on the design of software tools for detecting nondeterminism in parallel programs. We are attacking this problem at the source language level and are focusing on Cedar Fortran, which can be explicitly parallel, but the approach is equally applicable to other procedural languages.

The problem of detecting nondeterminacy is essentially a two–part problem. The first part is to analyze the flow of control a program takes when it executes, in order to build a flow graph, taking into account synchronization statements or constructs. The second part is to examine all the arcs in the graph, searching for unsynchronized accesses to the same variable. Such conflicting accesses constitute a *race* and are the cause of nondeterminism. This part of the process (race detection) is fairly straightforward and a number of researchers have been attacking this problem with success. The first part, identifying and analyzing the interrelationships of various types of synchronization, both explicit and implicit, is a much more difficult problem.

Prototype programs have been built for instrumenting Fortran programs in order to collect trace data, and to analyze this data in order to detect races. These tools have been shown to work on test programs that have a variety of races, but very limited types of synchronization patterns. We also have developed algorithms for analyzing parallel programs that use conventional *post*, *wait*, and *clear* synchronization primitives. That is, given the trace of a program, we can automatically associate posts with waits in parallel threads of the program. This lets us add synchronization arcs to the flow graph, leading to more accurate race detection.

We are still at an early stage in the development of these tools, but the results that we (and other researchers) have obtained are very encouraging. In order to provide practical tools, we are working at refining the user interface, particularly the reporting of analysis results. Also, we are planning to extend the set of synchronization primitives that can be handled, such as semaphores, *lock* and *unlock*, *advance* and *await*, and so on. Finally, we expect to apply similar techniques to static analysis, ideally leading to compilers which do automatic nondeterminacy detection.

Automatic Compiler Synchronization Generation

Current plans call for the Cedar Fortran compiler to generate synchronization instructions by September 1989. Software to experimentally determine the effectiveness of the synchronization optimization scheme of Midkiff and Padua [Midk86, MiPa86, MiPa87] is being developed, and results are expected by the second quarter. Research continues to extend these optimizations to loop nests with nested parallelism, and to synchronize cross loop dependences.

There are currently no plans to implement the synchronization scheme of Zhu and Yew [ZhYe84, ZhYe87] for loops containing subscripted subscripts. Experiments are being planned to determine the effectiveness of their method, and the extensions to that method developed by Midkiff and Padua. The final decision to implement will be based largely on the outcome of those experiments.

Error Analysis

Our efforts in the past have been centered around the generation of a robust and efficient pass to the Cedar Fortran preprocessor. The original purpose of this pass was to instrument the Fortran code so that when compiled and run it produces a trace appropriate for input to Larson's roundoff error-analysis package. The current version, however, is much more flexible: without any options, it converts expressions in the program to triad form, a task useful in its own right as it simplifies various other preprocessing passes. By specifying different options, it allows the instrumented program to perform a variety of functions:

- counting floating-point operations
- error-analysis using statistical methods by introducing controlled perturbations to operations.
- interval arithmetic
- generating a program trace of the floating-point computation that it was originally designed to do.

The preprocessor is in a state of completion that allows it to perform the first two functions on standard Fortran 77 programs, and the libraries with which the compiled code must be linked to implement them have been written. We have thus started using the preprocessor for performing statistical error analysis of numerical algorithms.

Unlike other floating-point operation count tools, our preprocessor does not disable the vectorization within a program, and returns a report of the program's approximate number of vector floating-point operations.

The implementation of the remaining preprocessor functions should be completed by July 1989.

Automatic Generation of Parallel PDE Solvers

The area of interest is to apply a symbolic algebra system as a tool for the manipulation of Partial Differential Equations (PDEs) into a form suitable for numerical solution on Cedar. The intended goal of this project is to evaluate the use of problem specifications as a higher-level method for the generation of parallel programs, and to provide a workstation interface to a symbolic algebra subsystem as the front-end for editing, simplification, and code generation of the PDEs. Preliminary work is with one- and two-dimensional PDEs over a rectangular domain.

The symbolic algebra program is remotely executed on the host computer. An algebra server was written as a translator for the symbolic algebra program. Routines have been developed to interact with the remote system in a uniform manner. The routines use a variation of the Lisp s-expression as the exchange format to remove binary incompatibilities and to allow a concise representation of the expression parse tree. The current instance of this is an interface to Macsyma configured to communicate over

TCP/IP to a SUN or VAX using the remote execution service. It simulates a normal terminal session to the symbolic algebra program by using the syntax required by the program while using a consistent parse tree representation for communication to the client process.

PDE–Tutor, based on work done by Wirth [Wirt80], was written to interactively specify the problem parameters. The user can specify the dimensions of the region, and the number of divisions to use in discretizing the problem. The interface to the symbolic algebra server is through a set of global variables defining the problem, region, boundary, and initial conditions that the user had specified in PDE–Tutor. The interface variables can then be used by several user–specified driving routines in Macsyma to process the equations by various methods and solution techniques.

Another aspect of the project is the user interface. The user interface is evolving as a graphical representation of the symbolic algebra server. The X Window System was chosen as the basis for the user interface for the reasons of portability to other workstations and to allow integration into the Faust project.

The user interface allows interaction with the algebra server and the display of the results in two–dimensional graphical form. The user interface is divided into three main areas: input, output, and selection. Additional tools are available to view the results of each phase in the translation process; for example, it is possible to view the coefficient matrix used in the implicit methods. The input area is implemented as a page editor for textual commands. These commands are then parsed into the internal representation and sent to the algebra server.

The output area is organized as a series of browser windows. The browsers are sliding windows onto the previous commands to and responses from the algebra server. These browsers allow the user to independently scroll through the list of equations and to interact with the equations. Commands are available to allow manipulation of the equations in the browser windows. These are to "collapse" and "expand" subexpressions, and to transfer selected subexpressions to the user's input buffer.

The selection area of the PDE–Tutor has been designed to be a graphical outline of the rectangular domain. Each subregion in the domain may be controlled by an independent equation for the initial, boundary, and problem specifications. Selecting the menu for a subregion will produce the selections for dealing with the equations in that subregion.

Subroutine Librarian

A prototype of an expert system librarian was completed. Using GPSI (General Purpose System for Inferencing), an expert system generator created by M. T. Harandi in the Department of Computer Science, a rule base was constructed to allow application programmers to locate optimized routines in the CSRD library. This work features an interactive textual interface that queried the user for appropriate information, and also gives the user the ability to have the system automatically ascertain certain information such as matrix attributes for unknown cases.

A second prototype for on-line documentation service was developed, based on a collaborative effort with Ames Laboratory. The system is a graphical documentation browser, and based on the Ames SLADOC system. Ames modified SLADOC to run on X Windows to allow this integration to take place. The second prototype features a more intuitive graphic user interface that is extendible by the user. Additionally, the interface allows for the retrieval of performance data to be rendered in graphical form so that the user may gain some insight into the expected behavior of the library routine when embedded into an application.

The database for this second prototype has been completed and is being evaluated by the CSRD applications group. The system features a hierarchical database browser and graphical database editor. The applications group is currently building the database to reflect the current state of the CSRD numerical libraries. We expect this librarian to be in general use by the end of the contract year.